1 [COM4513-6513] Assignment 2: Text Classification with a Feedforward Network

1.0.1 Instructor: Nikos Aletras

The goal of this assignment is to develop a Feedforward network for text classification.

For that purpose, you will implement:

- Text processing methods for transforming raw text data into input vectors for your network (**1 mark**)
- A Feedforward network consisting of:
  - **One-hot** input layer mapping words into an **Embedding weight matrix** (**1 mark**)
  - **One hidden layer** computing the mean embedding vector of all words in input followed by a **ReLU activation function** (**1 mark**)
  - **Output layer** with a **softmax** activation. (**1 mark**)
- The Stochastic Gradient Descent (SGD) algorithm with **back-propagation** to learn the weights of your Neural network. Your algorithm should:
  - Use (and minimise) the **Categorical Cross-entropy loss** function (**1 mark**)
  - Perform a **Forward pass** to compute intermediate outputs (**4 marks**)
  - Perform a **Backward pass** to compute gradients and update all sets of weights (**4 marks**)
  - Implement and use **Dropout** after each hidden layer for regularisation (**2 marks**)
- Discuss how did you choose hyperparameters? You can tune the learning rate (hint: choose small values), embedding size {e.g. 50, 300, 500}, the dropout rate {e.g. 0.2, 0.5} and the learning rate. Please use tables or graphs to show training and validation

performance for each hyperparam combination (**2 marks**).

- After training the model, plot the learning process (i.e. training and validation loss in each epoch) using a line plot and report accuracy.
- Re-train your network by using pre-trained embeddings (GloVe) trained on large corpora. Instead of randomly initialising the embedding weights matrix, you should initialise it with the pre-trained weights. During training, you should not update them (i.e. weight freezing) and backprop should stop before computing gradients for updating embedding weights. Report results by performing hyperparameter tuning and plotting the learning process. Do you get better performance? (**3 marks**).
- **BONUS:** Extend you Feedforward network by adding more hidden layers (e.g. one more). How does it affect the performance? Note: You need to repeat hyperparameter tuning, but the number of combinations grows exponentially. Therefore, you need to choose a subset of all possible combinations (**+2 extra marks**)

1.0.2 Data

The data you will use for Task 2 is a subset of the AG News Corpus and you can find it in the `./data_topic` folder in CSV format:

- `data_topic/train.csv`: contains 2,400 news articles, 800 for each class to be used for training.
- `data_topic/dev.csv`: contains 150 news articles, 50 for each class to be used for hyperparameter selection and monitoring the training process.
- `data_topic/test.csv`: contains 900 news articles, 300 for each class to be used for testing.

1.0.3 Pre-trained Embeddings

You can download pre-trained GloVe embeddings trained on Common Crawl (840B tokens, 2.2M vocab, cased, 300d vectors, 2.03 GB download) from here. No need to unzip, the file is large.

1.0.4 Save Memory

To save RAM, when you finish each experiment you can delete the weights of your network using `del W` followed by Python' s garbage collector `gc.collect()`

1.0.5 Submission Instructions

You should submit a Jupyter Notebook file (assignment2.ipynb) and an exported PDF version (you can do it from Jupyter: `File->Download as->PDF via Latex`).

You are advised to follow the code structure given in this notebook by completing all given funtions. You can also write any auxilliary/helper functions (and arguments for the functions) that you might need but note that you can provide a full solution without any such functions. Similarly, you can just use only the packages imported below but you are free to use any functionality from the Python Standard Library, NumPy, SciPy and Pandas. You are not allowed to use any third-party library such as Scikit-learn (apart from metric functions already provided), NLTK, Spacy, Keras etc.. You are allowed to re-use your code from Assignment 1.

Please make sure to comment your code. You should also mention if you' ve used Windows to write and test your code. There is no single correct answer on what your accuracy should be, but correct implementations usually achieve F1 of ~75-80% and ~85% without and with using pre-trained embeddings respectively.

This assignment will be marked out of 20. It is worth 20% of your final grade in the module. If you implement the bonus question you can get up to 2 extra points but your final grade will be capped at 20.

The deadline for this assignment is **23:59 on Mon, 18 May 2020** and it needs to be submitted via Blackboard (MOLE). Standard departmental penalties for lateness will be applied. We use a range of strategies to detect unfair means, including Turnitin which helps detect plagiarism, so make sure you do not plagiarise.

[1]:
```
import pandas as pd
import numpy as np
from collections import Counter
import re
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
↪f1_score
import random
from time import localtime, strftime
from scipy.stats import spearmanr,pearsonr
import zipfile
```

```
import gc

# fixing random seed for reproducibility
random.seed(123)
np.random.seed(123)
```

1.1 Transform Raw texts into training and development data

First, you need to load the training, development and test sets from their corresponding CSV files (tip: you can use Pandas dataframes).

[2]:
```
train_data = pd.read_csv("./data_topic/train.
 ↪csv",header=None,names=["label","text"])
dev_data = pd.read_csv("./data_topic/dev.
 ↪csv",header=None,names=["label","text"])
test_data = pd.read_csv("./data_topic/test.
 ↪csv",header=None,names=["label","text"])
```

[3]:
```
# Output the demo of the text
train_data[:10]
```

[3]:

	label	text
0	1	Reuters - Venezuelans turned out early\and in …
1	1	Reuters - South Korean police used water canno…
2	1	Reuters - Thousands of Palestinian\prisoners i…
3	1	AFP - Sporadic gunfire and shelling took place…
4	1	AP - Dozens of Rwandan soldiers flew into Suda…
5	1	Reuters - Rwandan troops were airlifted on Sun…
6	1	AP - A bomb exploded during an Independence Da…
7	1	AFP - Australia's foreign minister will pay a …
8	1	AP - Democratic presidential candidate John Ke…
9	1	AP - Democratic vice presidential candidate Jo…

[4]:
```
# Divide the pdframe into label list and text list
x_tr = train_data['text'].tolist()
x_label_tr = np.array(train_data['label'])
# development dataset
```

```
x_dev = dev_data['text'].tolist()
x_label_dev = np.array(dev_data['label'])
# test dataset
x_test = test_data['text'].tolist()
x_label_test = np.array(test_data['label'])
```

[5]:
```
# Create a function to convert the uppercase to lowercase
def upper_tolower(array):
    lower_list = list()
    for i in range(len(array)):
        lower_list.append(array[i].lower())
    return lower_list

x_tr = upper_tolower(x_tr)
x_dev = upper_tolower(x_dev)
x_test = upper_tolower(x_test)
```

2 Create input representations

To train your Feedforward network, you first need to obtain input representations given a vocabulary. One-hot encoding requires large memory capacity. Therefore, we will instead represent documents as lists of vocabulary indices (each word corresponds to a vocabulary index).

2.1 Text Pre-Processing Pipeline

To obtain a vocabulary of words. You should: - tokenise all texts into a list of unigrams (tip: you can re-use the functions from Assignment 1) - remove stop words (using the one provided or one of your preference) - remove unigrams appearing in less than K documents - use the remaining to create a vocabulary of the top-N most frequent unigrams in the entire corpus.

[6]:
```
stop_words = ['a','in','on','at','and','or',
              'to', 'the', 'of', 'an', 'by',
              'as', 'is', 'was', 'were', 'been', 'be',
              'are','for', 'this', 'that', 'these', 'those', 'you', 'i', 'if',
             'it', 'he', 'she', 'we', 'they', 'will', 'have', 'has',
```

```
            'do', 'did', 'can', 'could', 'who', 'which', 'what',
            'but', 'not', 'there', 'no', 'does', 'not', 'so', 've', 'their',
           'his', 'her', 'they', 'them', 'from', 'with', 'its']
```

2.1.1 Unigram extraction from a document

You first need to implement the `extract_ngrams` function. It takes as input: - `x_raw`: a string corresponding to the raw text of a document - `ngram_range`: a tuple of two integers denoting the type of ngrams you want to extract, e.g. (1,2) denotes extracting unigrams and bigrams. - `token_pattern`: a string to be used within a regular expression to extract all tokens. Note that data is already tokenised so you could opt for a simple white space tokenisation. - `stop_words`: a list of stop words - `vocab`: a given vocabulary. It should be used to extract specific features.

and returns:

- a list of all extracted features.

[7]:
```
def extract_ngrams(x_raw, ngram_range=(1,3),␣
↪token_pattern=r'\b[A-Za-z][A-Za-z]+\b', stop_words=[], vocab=set()):
    # Next it is my part to finish the this part
    # Create a new blank list
    x = list()
    ngram_list = list()

    for x_each in x_raw:
        x_findword = re.findall(token_pattern,x_each)
        # Now we have obtained the new word list without the stop_word
        x_rawall = [word for word in x_findword if word not in stop_words]
        store_ngram = list()
        for n_g in range(min(ngram_range),max(ngram_range)+1):
            for i in range(len(x_rawall)-n_g+1):
                if n_g == 1:
                    ngramTemp = x_rawall[i]
                    x.append(ngramTemp)
                    store_ngram.append(ngramTemp)
                elif n_g == 2 or n_g == 3:
                    ngramTemp = tuple(x_rawall[i:i+n_g])
```

```
                    x.append(ngramTemp)
                    store_ngram.append(ngramTemp)
        ngram_list.append(store_ngram)

    return ngram_list
```

2.1.2 Create a vocabulary of n-grams

Then the `get_vocab` function will be used to (1) create a vocabulary of ngrams; (2) count the document frequencies of ngrams; (3) their raw frequency. It takes as input: - `X_raw`: a list of strings each corresponding to the raw text of a document - `ngram_range`: a tuple of two integers denoting the type of ngrams you want to extract, e.g. (1,2) denotes extracting unigrams and bigrams. - `token_pattern`: a string to be used within a regular expression to extract all tokens. Note that data is already tokenised so you could opt for a simple white space tokenisation. - `stop_words`: a list of stop words - `min_df`: keep ngrams with a minimum document frequency. - `keep_topN`: keep top-N more frequent ngrams.

and returns:

- `vocab`: a set of the n-grams that will be used as features.
- `df`: a Counter (or dict) that contains ngrams as keys and their corresponding document frequency as values.
- `ngram_counts`: counts of each ngram in vocab

[8]:
```
def get_vocab(X_raw, ngram_range=(1,3),␣
↪token_pattern=r'\b[A-Za-z][A-Za-z]+\b',min_df=0, keep_topN=0, stop_words=[]):

    df = dict() #dict that contains ngrams as keys and their corresponding␣
↪document frequency as values.
    ngram_counts= list() # type is the list and stores the number of words␣
↪which occur at the text

    # only obtain the unigram
    vocab = extract_ngrams(X_raw, ngram_range = ngram_range,␣
↪stop_words=stop_words)

    for each_ngram in vocab:
```

```
        for ngram in set(each_ngram):
            if ngram not in df.keys():
                df[ngram] = 1
            else:
                df[ngram] +=1

    df = dict(sorted(df.items(), key=lambda count:count[1], reverse=True)[:
↪keep_topN])

    vocab = [name for name in df.keys()]

    for item in df.values():
        ngram_counts.append(item)

    return vocab, df, ngram_counts
```

Now you should use `get_vocab` to create your vocabulary and get document and raw frequencies of unigrams:

[9]:
```
vocab,df,ngram_counts = get_vocab(x_tr, ngram_range=(1,1), keep_topN=2000,␣
↪stop_words=stop_words)
vocab_dev, df_dev, ngram_counts_dev = get_vocab(x_dev, ngram_range=(1,1),␣
↪keep_topN=2000, stop_words=stop_words)
vocab_test, df_test, ngram_counts_test = get_vocab(x_test, ngram_range=(1,1),␣
↪keep_topN=2000, stop_words=stop_words)
```

[10]:
```
print(len(vocab))
print()
print(list(vocab)[:100])
print()
print(list(df.items())[:10])
```

```
2000

['reuters', 'said', 'tuesday', 'wednesday', 'new', 'after', 'ap', 'athens',
'monday', 'first', 'two', 'york', 'over', 'us', 'olympic', 'inc', 'more',
'year', 'oil', 'prices', 'company', 'world', 'than', 'aug', 'about', 'had',
```

```
'united', 'one', 'sunday', 'out', 'into', 'against', 'up', 'second', 'last',
'president', 'stocks', 'gold', 'team', 'when', 'three', 'night', 'time',
'yesterday', 'games', 'olympics', 'states', 'greece', 'off', 'iraq',
'washington', 'percent', 'home', 'day', 'google', 'public', 'record', 'week',
'men', 'government', 'win', 'american', 'won', 'years', 'all', 'billion',
'shares', 'city', 'offering', 'officials', 'would', 'today', 'final', 'afp',
'gt', 'people', 'lt', 'medal', 'corp', 'sales', 'country', 'back', 'four',
'high', 'investor', 'com', 'minister', 'reported', 'month', 'initial', 'profit',
'ticker', 'al', 'million', 'top', 'before', 'china', 'him', 'national', 'najaf']

[('reuters', 631), ('said', 432), ('tuesday', 413), ('wednesday', 344), ('new',
325), ('after', 295), ('ap', 275), ('athens', 245), ('monday', 221), ('first',
210)]
```

Then, you need to create vocabulary id -> word and id -> word dictionaries for reference:

[11]:
```
def create_dict(vocab):
    vocab_word_id = dict()
    for i in range(len(vocab)):
        vocab_word_id[vocab[i]] = i
    return vocab_word_id
```

[12]:
```
vocab_word_id = create_dict(vocab)
```

2.1.3 Convert the list of unigrams into a list of vocabulary indices

Storing actual one-hot vectors into memory for all words in the entire data set is prohibitive. Instead, we will store word indices in the vocabulary and look-up the weight matrix. This is equivalent of doing a dot product between an one-hot vector and the weight matrix.

First, represent documents in train, dev and test sets as lists of words in the vocabulary:

[13]:
```
def convert_indices(vocab,vocab_word_id):
    vocab_indices = list()
    X_uni_tr = vocab
    for sid in range(len(X_uni_tr)):
        temp_list = list()
        for word in X_uni_tr[sid]:
            if word in vocab_word_id:
```

```
                id_tag = vocab_word_id[word]
                temp_list.append(id_tag)
        vocab_indices.append(temp_list)
    return X_uni_tr, vocab_indices
```

[14]:
```
vocab_tr = extract_ngrams(x_tr, ngram_range=(1,1),stop_words=stop_words)
vocab_dev = extract_ngrams(x_dev, ngram_range=(1,1),stop_words=stop_words)
vocab_test = extract_ngrams(x_test, ngram_range=(1,1),stop_words=stop_words)
```

[15]:
```
X_uni_tr,X_tr = convert_indices(vocab_tr,vocab_word_id)
X_uni_dev,X_dev = convert_indices(vocab_dev,vocab_word_id)
X_uni_test,X_test = convert_indices(vocab_test,vocab_word_id)
```

[16]:
```
# In the last section We should delete the test set which could obtain the null␣
 ↪value for the articles
def delete_null(data,label):
    X_true = list()
    Y_true = list()
    for i in range(len(data)):
        if len(data[i])==0:
            continue
        X_true.append(data[i])
        Y_true.append(label[i])
    Y_true = np.array(Y_true)
    return X_true,Y_true
```

[17]:
```
X_tr,x_label_tr = delete_null(X_tr,x_label_tr)
X_dev,x_label_dev = delete_null(X_dev,x_label_dev)
X_test,x_label_test = delete_null(X_test,x_label_test)
```

[18]:
```
X_uni_tr[0]
```

[18]:
```
['reuters',
 'venezuelans',
 'turned',
 'out',
 'early',
 'large',
```

```
 'numbers',
 'sunday',
 'vote',
 'historic',
 'referendum',
 'either',
 'remove',
 'left',
 'wing',
 'president',
 'hugo',
 'chavez',
 'office',
 'give',
 'him',
 'new',
 'mandate',
 'govern',
 'next',
 'two',
 'years']
```

Then convert them into lists of indices in the vocabulary:

[19]:
```
X_tr[0]
```

[19]:
```
[0,
 1011,
 758,
 29,
 208,
 1103,
 1367,
 28,
 308,
 816,
 262,
```

```
1586,
108,
759,
35,
172,
175,
493,
701,
97,
4,
1221,
173,
10,
63]
```

3 Network Architecture

Your network should pass each word index into its corresponding embedding by looking-up on the embedding matrix and then compute the first hidden layer $\mathbf{h}_1$:

$$\mathbf{h}_1 = \frac{1}{|x|} \sum_i W_i^e, i \in x$$

where $|x|$ is the number of words in the document and W^e is an embedding matrix $|V| \times d$, $|V|$ is the size of the vocabulary and d the embedding size.

Then $\mathbf{h}_1$ should be passed through a ReLU activation function:

$$\mathbf{a}_1 = relu(\mathbf{h}_1)$$

Finally the hidden layer is passed to the output layer:

$$\mathbf{y} = \text{softmax}(\mathbf{a}_1 W^T)$$

where W is a matrix $d \times |\mathcal{Y}|$, $|\mathcal{Y}|$ is the number of classes.

During training, $\mathbf{a}_1$ should be multiplied with a dropout mask vector (elementwise) for regularisation before it is passed to the output layer.

You can extend to a deeper architecture by passing a hidden layer to another one:

$$\mathbf{h_i} = \mathbf{a}_{i-1} W_i^T$$

$$\mathbf{a_i} = relu(\mathbf{h_i})$$

4 Network Training

First we need to define the parameters of our network by initiliasing the weight matrices. For that purpose, you should implement the `network_weights` function that takes as input:

- `vocab_size`: the size of the vocabulary
- `embedding_dim`: the size of the word embeddings
- `hidden_dim`: a list of the sizes of any subsequent hidden layers (for the Bonus). Empty if there are no hidden layers between the average embedding and the output layer
- `num_clusses`: the number of the classes for the output layer

and returns:

- `W`: a dictionary mapping from layer index (e.g. 0 for the embedding matrix) to the corresponding weight matrix initialised with small random numbers (hint: use numpy.random.uniform with from -0.1 to 0.1)

See the examples below for expected outputs. Make sure that the dimensionality of each weight matrix is compatible with the previous and next weight matrix, otherwise you won't be able to perform forward and backward passes. Consider also using np.float32 precision to save memory.

[20]:
```
def network_weights(vocab_size=1000, embedding_dim=300,
                    hidden_dim=[], num_classes=3, init_val = 0.5):

    # Initialize the first layer for the input
    # Firstly we should initilize the list
    layer_num_list = list()
    # Insert the hidden layer
```

```
    layer_num_list.extend(hidden_dim)
    # Insert vocab_size
    layer_num_list.insert(0,vocab_size)
    layer_num_list.insert(1,embedding_dim)
    layer_num_list.append(num_classes)
    W = dict()
    for sid in range(len(layer_num_list)):
        if sid == (len(layer_num_list)-1):
            break
        else:
            W[sid] = np.random.
↪uniform(-init_val,init_val,(layer_num_list[sid],layer_num_list[sid+1])).
↪astype("float32")
    return W
```

[21]:
```
W = network_weights(vocab_size=5,embedding_dim=10,hidden_dim=[], num_classes=2)

print('W_emb:', W[0].shape)
print('W_out:', W[1].shape)
```

```
W_emb: (5, 10)
W_out: (10, 2)
```

[22]:
```
W = network_weights(vocab_size=3,embedding_dim=4,hidden_dim=[2], num_classes=2)
```

[23]:
```
print('W_emb:', W[0].shape)
print('W_h1:', W[1].shape)
print('W_out:', W[2].shape)
```

```
W_emb: (3, 4)
W_h1: (4, 2)
W_out: (2, 2)
```

[24]:
```
W[0]
```

[24]:
```
array([[-0.4042875 ,  0.38532683,  0.12724897,  0.22341636],
       [-0.4838708 ,  0.09443188,  0.05678519, -0.34104034],
       [-0.3469295 ,  0.19552954, -0.18123357,  0.19197029]],
```

```
      dtype=float32)
```

Then you need to develop a `softmax` function (same as in Assignment 1) to be used in the output layer. It takes as input:

- `z`: array of real numbers

and returns:

- `sig`: the softmax of `z`

[25]:
```
def softmax(z):
    x_result = 0.0
    if len(z.shape)>1:
        x_result = np.sum(np.exp(z), axis = 1, keepdims = True)
    else:
        x_result = np.sum(np.exp(z), axis = 0)
    result = np.exp(z) / x_result
    return result
```

Now you need to implement the categorical cross entropy loss by slightly modifying the function from Assignment 1 to depend only on the true label `y` and the class probabilities vector `y_preds`:

[26]:
```
def categorical_loss(y, y_preds):
    l = -np.log(y_preds[y])
    return l
```

[27]:
```
# example for 5 classes
y = 2 #true label
y_preds = softmax(np.array([[-2.1,1.,0.9,-1.3,1.5]]))[0]

print('y_preds: ',y_preds)
print('loss:', categorical_loss(y, y_preds))
```

```
y_preds:  [0.01217919 0.27035308 0.24462558 0.02710529 0.44573687]
loss: 1.40802648485675
```

Then, implement the `relu` function to introduce non-linearity after each hidden layer of your network (during the forward pass):

$$relu(z_i) = max(z_i, 0)$$

and the `relu_derivative` function to compute its derivative (used in the backward pass):

$$\text{relu}_derivative\text{relu}_derivative\text{relu}_derivative\text{relu}_derivative(z_i) = \begin{cases} 0, & \text{if } z_i <= 0. \\ 1, & \text{otherwise.} \end{cases} \quad (1)$$

Note that both functions take as input a vector z

Hint use .copy() to avoid in place changes in array z

[28]:
```
def relu(z):
    # use the numpy maxium function to select the z value
    result = np.maximum(z,0)
    return result

def relu_derivative(z):
    result = np.array(z)
    result[result<=0]=0
    result[result>0]=1
    return result
```

During training you should also apply a dropout mask element-wise after the activation function (i.e. vector of ones with a random percentage set to zero). The `dropout_mask` function takes as input:

- `size`: the size of the vector that we want to apply dropout
- `dropout_rate`: the percentage of elements that will be randomly set to zeros

and returns:

- `dropout_vec`: a vector with binary values (0 or 1)

[29]:
```
def dropout_mask(size, dropout_rate):
    array = np.ones(size).astype("float_")
    array[:int(size*dropout_rate)] = 0.0
    np.random.shuffle(array)
    return array
```

[30]:
```
print(dropout_mask(10, 0.2))
print(dropout_mask(10, 0.2))
```

```
[1. 1. 1. 1. 0. 1. 0. 1. 1. 1.]
[1. 1. 1. 1. 1. 0. 1. 0. 1. 1.]
```

Now you need to implement the `forward_pass` function that passes the input x through the network up to the output layer for computing the probability for each class using the weight matrices in `W`. The ReLU activation function should be applied on each hidden layer.

- `x`: a list of vocabulary indices each corresponding to a word in the document (input)
- `W`: a list of weight matrices connecting each part of the network, e.g. for a network with a hidden and an output layer: W[0] is the weight matrix that connects the input to the first hidden layer, W[1] is the weight matrix that connects the hidden layer to the output layer.
- `dropout_rate`: the dropout rate that is used to generate a random dropout mask vector applied after each hidden layer for regularisation.

and returns:

- `out_vals`: a dictionary of output values from each layer: h (the vector before the activation function), a (the resulting vector after passing h from the activation function), its dropout mask vector; and the prediction vector (probability for each class) from the output layer.

[31]:
```
# Now We could obtain the forward_pass function
def forward_pass(x, W, dropout_rate=0.2):

    out_vals = {}
    h_vecs = []
    a_vecs = []
    dropout_vecs = []

    W_length = len(W)-1
    x_vector = list()
    for i in x:
        x_vector.append(W[0][i])

    # Initialize the function
```

```
    h = np.sum(x_vector,axis=0)/len(x)
    a = relu(h)
    d = dropout_mask(len(a),dropout_rate)
    output = a*d
    h_vecs.append(h)
    a_vecs.append(a)
    dropout_vecs.append(d)

    for i in range(W_length):
        if i == 0:
            continue
        else:
            h = np.dot(output,W[i])
        # Update the h value
        a = relu(h)
        d = dropout_mask(len(a) ,dropout_rate)
        output = a*d
        h_vecs.append(h)
        a_vecs.append(a)
        dropout_vecs.append(d)

    # USe the softmax to obtain the y value
    y_array = softmax(np.dot(output,W[W_length]))

    out_vals['h'] = h_vecs
    out_vals['a'] = a_vecs
    out_vals['dropout_vecs'] = dropout_vecs
    out_vals['y'] = y_array

    return out_vals
```

[32]:
```
W = network_weights(vocab_size=3,embedding_dim=4,hidden_dim=[5], num_classes=2)

for i in range(len(W)):
    print('Shape W'+str(i), W[i].shape)
```

```
print()
print(forward_pass([2,1], W, dropout_rate=0.5))
```

```
Shape W0 (3, 4)
Shape W1 (4, 5)
Shape W2 (5, 2)

{'h': [array([-0.04668263, -0.12518334,  0.17532286, -0.32932055],
dtype=float32), array([-0.01851934,  0.04051239, -0.05942235,  0.01765476,
0.0641444 ])], 'a': [array([0.        , 0.        , 0.17532286, 0.        ],
dtype=float32), array([0.        , 0.04051239, 0.        , 0.01765476, 0.0641444
])], 'dropout_vecs': [array([1., 0., 1., 0.]), array([0., 0., 1., 1., 1.])],
'y': array([0.48946731, 0.51053269])}
```

The `backward_pass` function computes the gradients and update the weights for each matrix in the network from the output to the input. It takes as input

- `x`: a list of vocabulary indices each corresponding to a word in the document (input)
- `y`: the true label
- `W`: a list of weight matrices connecting each part of the network, e.g. for a network with a hidden and an output layer: W[0] is the weight matrix that connects the input to the first hidden layer, W[1] is the weight matrix that connects the hidden layer to the output layer.
- `out_vals`: a dictionary of output values from a forward pass.
- `learning_rate`: the learning rate for updating the weights.
- `freeze_emb`: boolean value indicating whether the embedding weights will be updated.

and returns:

- `W`: the updated weights of the network.

Hint: the gradients on the output layer are similar to the multiclass logistic regression.

[33]:
```
# Now we should imply the function backware_pass function
def backward_pass(x, y, W, out_vals, lr=0.001, freeze_emb=False):
    W_length = len(W)-1
    for i in range(W_length):
        if i == 0:
```

```
            y_layer = np.zeros(W[W_length].shape[1])
            y_layer[y-1]=1
            temp = out_vals['y'] - y_layer
            output_v1 = out_vals['a'][-1]*out_vals['dropout_vecs'][-1]
            output_value = output_v1.reshape(W[W_length].shape[0],1)
            # Obtain the gradient value
            gradient_value = np.dot(output_value,temp.reshape(1,W[W_length].
↪shape[1]))
            # Update the temp for layer
            v1 = np.dot(W[W_length],temp).reshape(1,W[W_length].shape[0])
            v2 = out_vals['dropout_vecs'][W_length-1]
            temp = v1*v2
            # Update the Weight Matrix
            W[W_length] = W[W_length] - lr*gradient_value


        else:
            # Obtain the value of function relu_derivative function to compute␣
↪its derivative
            der_v = relu_derivative(out_vals['h'][W_length-i])
            # Update the temp value
            temp = temp * der_v.reshape(1,W[W_length+1-i].shape[0])
            # Next Calculate the output value
            output_v1 =␣
↪out_vals['a'][W_length-1-i]*out_vals['dropout_vecs'][W_length-1-i]
            output_value = output_v1.reshape(W[W_length-i].shape[0],1)
            # Obtain the gradient vaalue
            gradient_value = np.dot(output_value,temp)
            # Update the temp
            temp_v1 = np.dot(W[W_length-i],temp.T).reshape(1,W[W_length-i].
↪shape[0])
            temp_v2 = out_vals['dropout_vecs'][W_length-1-i]
            temp = temp_v1*temp_v2
            # Calculate the new W
            W[W_length-i] = W[W_length-i] - lr*gradient_value
```

```
    # Update the W0 if freeze_emb==false
    if not freeze_emb:
        x_array = np.zeros([W[0].shape[0],1])
        x_array[x] = 1.0
        lv_1 = relu_derivative(out_vals['h'][0]).reshape(1,W[0].shape[1])
        temp = temp*lv_1
        w_gradient = np.dot(x_array,temp)
        W[0] = W[0]-lr*w_gradient

    return W
```

Finally you need to modify SGD to support back-propagation by using the `forward_pass` and `backward_pass` functions.

The `SGD` function takes as input:

- `X_tr`: array of training data (vectors)
- `Y_tr`: labels of `X_tr`
- `W`: the weights of the network (dictionary)
- `X_dev`: array of development (i.e. validation) data (vectors)
- `Y_dev`: labels of `X_dev`
- `lr`: learning rate
- `dropout`: regularisation strength
- `epochs`: number of full passes over the training data
- `tolerance`: stop training if the difference between the current and previous validation loss is smaller than a threshold
- `freeze_emb`: boolean value indicating whether the embedding weights will be updated (to be used by the backward pass function).
- `print_progress`: flag for printing the training progress (train/validation loss)

and returns:

- `weights`: the weights learned
- `training_loss_history`: an array with the average losses of the whole training set after each epoch
- `validation_loss_history`: an array with the average losses of the whole development set after each epoch

[34]:
```
# Obtain the average loss
def loss_function(x, y, W, dropout_rate):
    result = 0.0
    for sid in range(len(x)):
        out_vals = forward_pass(x[sid], W, dropout_rate)
        y_label = out_vals['y']
        loss_v = categorical_loss(y[sid]-1,y_label)
        result += loss_v
    return result/(len(x))
```

[35]:
```
def SGD(X_tr, Y_tr, W, X_dev=[], Y_dev=[], lr=0.001,
        dropout=0.2, epochs=5, tolerance=0.001, freeze_emb=False,␣
↪print_progress=True):


    loss_dev = 10.0
    loss_tr_history = list()
    loss_vali_history = list()

    for sid in range(epochs):
        seed_num = random.randint(1,50)
        #Randomise the order of training data after each epoch
        np.random.seed(seed_num)
        random_X_tr = np.random.permutation(X_tr)
        np.random.seed(seed_num)
        random_Y_tr = np.random.permutation(Y_tr)

        # If the first is the first layer we should update the bw_result
        for i,tr_row in enumerate(random_X_tr):
            if i==0:
                fw_result = forward_pass(tr_row, W, dropout_rate=dropout)
                bw_result = backward_pass(tr_row, random_Y_tr[i], W, fw_result,␣
↪lr=lr, freeze_emb=freeze_emb)
            else:
                fw_result = forward_pass(tr_row, W, dropout_rate=dropout)
                bw_result = backward_pass(tr_row, random_Y_tr[i], bw_result,␣
↪fw_result, lr=lr, freeze_emb=freeze_emb)
```

```
        # Compute the train loss
        loss_train = loss_function(random_X_tr, random_Y_tr, bw_result,␣
↪dropout_rate=dropout)
        loss_tr_history.append(loss_train)

        #compute validation set loss
        loss_valid = loss_function(X_dev, Y_dev, bw_result,␣
↪dropout_rate=dropout)
        loss_vali_history.append(loss_valid)

        if print_progress:
            print('Epoch: %d' % sid, '| Training loss: %f' % loss_train, '|␣
↪Validation loss: %f' % loss_valid)

        if (loss_dev - loss_valid) < tolerance:
            break

        loss_dev = loss_valid

    return W, loss_tr_history, loss_vali_history
```

Now you are ready to train and evaluate you neural net. First, you need to define your network using the `network_weights` function followed by SGD with backprop:

[36]:
```
W = network_weights(vocab_size=len(vocab),embedding_dim=300,hidden_dim=[],␣
↪num_classes=3)

for i in range(len(W)):
    print('Shape W'+str(i), W[i].shape)

W, loss_tr, dev_loss = SGD(X_tr, Y_tr=x_label_tr,
                           W=W,
                           X_dev=X_dev,
                           Y_dev=x_label_dev,
                           lr=0.01,
```

```
                                dropout=0.6,
                                freeze_emb=False,
                                tolerance=0.0001,
                                epochs=30)
```

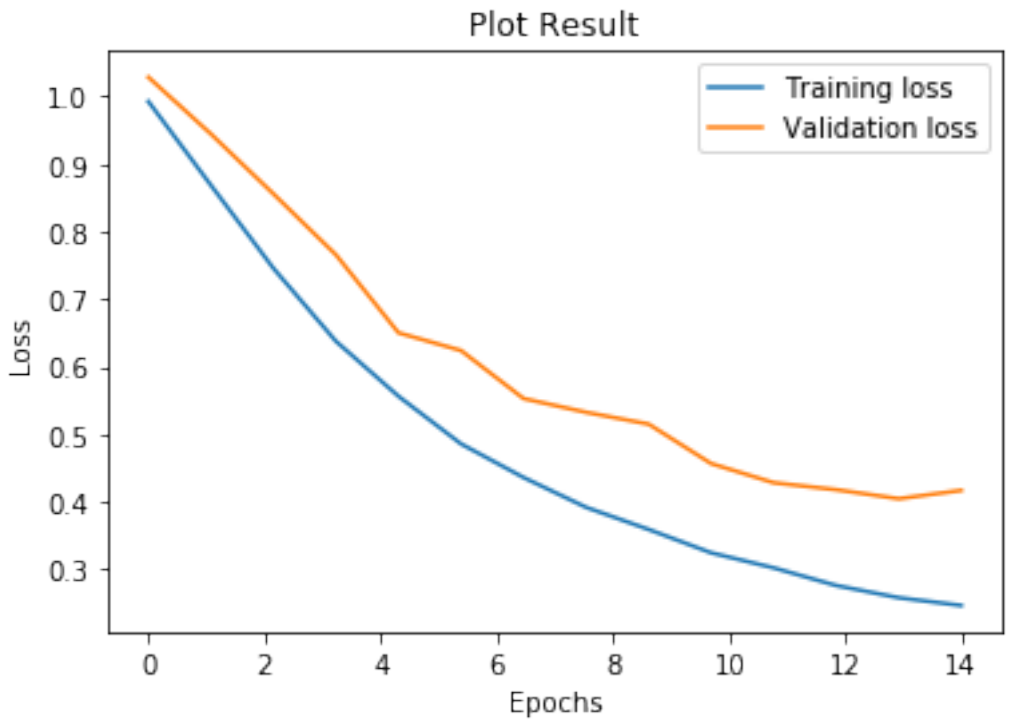
```
Shape W0 (2000, 300)
Shape W1 (300, 3)
Epoch: 0 | Training loss: 0.992532 | Validation loss: 1.028403
Epoch: 1 | Training loss: 0.868502 | Validation loss: 0.944363
Epoch: 2 | Training loss: 0.745471 | Validation loss: 0.855622
Epoch: 3 | Training loss: 0.637465 | Validation loss: 0.765498
Epoch: 4 | Training loss: 0.556079 | Validation loss: 0.649971
Epoch: 5 | Training loss: 0.485607 | Validation loss: 0.623784
Epoch: 6 | Training loss: 0.435832 | Validation loss: 0.552651
Epoch: 7 | Training loss: 0.391671 | Validation loss: 0.532314
Epoch: 8 | Training loss: 0.358932 | Validation loss: 0.514875
Epoch: 9 | Training loss: 0.324324 | Validation loss: 0.456412
Epoch: 10 | Training loss: 0.302069 | Validation loss: 0.428071
Epoch: 11 | Training loss: 0.275911 | Validation loss: 0.417748
Epoch: 12 | Training loss: 0.257846 | Validation loss: 0.404394
Epoch: 13 | Training loss: 0.246372 | Validation loss: 0.416714
```

Plot the learning process:

[37]:
```
# Now we could plot result of the training
x = np.linspace(0,len(loss_tr),len(loss_tr))
y1, y2 = loss_tr, dev_loss

plt.plot(x, y1,label='Training loss')
plt.plot(x, y2, label='Validation loss')

plt.title('Plot Result')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



Compute accuracy, precision, recall and F1-Score:

[38]:
```
preds_te = [np.argmax(forward_pass(x, W, dropout_rate=0.0)['y'])+1 for x,y in␣
 ↪zip(X_test,x_label_test)]
print('Accuracy:', accuracy_score(x_label_test,preds_te))
print('Precision:', precision_score(x_label_test,preds_te,average='macro'))
print('Recall:', recall_score(x_label_test,preds_te,average='macro'))
print('F1-Score:', f1_score(x_label_test,preds_te,average='macro'))
```

```
Accuracy: 0.8509454949944383
Precision: 0.8549264132513853
Recall: 0.8509438870308434
F1-Score: 0.8505701535931153
```

4.0.1 Discuss how did you choose model hyperparameters ?

[39]:
```
# Firstly We should establish the list to select the parameters
dim_choice=[200,300]
lr_rate = [0.01,0.001]
dropout_rate = [0.4,0.5]
hyper_result = 0.0
test_result = 0.0
result = list()

for dim in range(len(dim_choice)):
    for lr in range(len(lr_rate)):
        for drop in range(len(dropout_rate)):
            W =␣
↪network_weights(vocab_size=len(vocab),embedding_dim=dim_choice[dim],hidden_dim=[],␣
↪num_classes=3, init_val = 0.1)
            W, loss_tr, dev_loss = SGD(X_tr, Y_tr=x_label_tr,
                                W=W,
                                X_dev=X_dev,
                                Y_dev=x_label_dev,
                                lr=lr_rate[lr],
                                dropout=dropout_rate[drop],
                                freeze_emb=False,
                                tolerance=0.0001,
                                print_progress = False,
                                epochs=30)

            # Obtain the each result and store into different list
            # Firstly we should eveluate the performance of train
            preds_te = [np.argmax(forward_pass(x, W, dropout_rate=0.0)['y'])+1␣
↪for x,y in zip(X_tr,x_label_tr)]
            tr_f1 = f1_score(x_label_tr,preds_te,average='macro')
            hyper_result = tr_f1

            # Obtain the test set result
            preds_te = [np.argmax(forward_pass(x, W, dropout_rate=0.0)['y'])+1␣
↪for x,y in zip(X_test,x_label_test)]
```

```
            test_f1 = f1_score(x_label_test,preds_te,average='macro')
            test_result = test_f1

            result.append([hyper_result,test_result])
            print("emb_dim:"+str(dim_choice[dim])+" lr:"+str(lr_rate[lr])+"␣
↪drop_rate:"+str(dropout_rate[drop])+" The F1_score of Train and␣
↪Test"+str(result)+"\n")
            result = list()
```

```
emb_dim:200 lr:0.01 drop_rate:0.4 The F1_score of Train and
Test[[0.9542535833495561, 0.8483772786701067]]

emb_dim:200 lr:0.01 drop_rate:0.5 The F1_score of Train and
Test[[0.948799412115832, 0.8527132175044295]]

emb_dim:200 lr:0.001 drop_rate:0.4 The F1_score of Train and
Test[[0.7853065896886702, 0.7345424272146301]]

emb_dim:200 lr:0.001 drop_rate:0.5 The F1_score of Train and
Test[[0.3392127567652488, 0.27585050519044374]]

emb_dim:300 lr:0.01 drop_rate:0.4 The F1_score of Train and
Test[[0.9635434622519151, 0.855111346392309]]

emb_dim:300 lr:0.01 drop_rate:0.5 The F1_score of Train and
Test[[0.9364602288498646, 0.853923996631182]]

emb_dim:300 lr:0.001 drop_rate:0.4 The F1_score of Train and
Test[[0.6852044077625864, 0.6216461622860118]]

emb_dim:300 lr:0.001 drop_rate:0.5 The F1_score of Train and
Test[[0.7911975739050883, 0.738463835556189]]
```

5 Use Pre-trained Embeddings

Now re-train the network using GloVe pre-trained embeddings. You need to modify the `backward_pass` function above to stop computing gradients and updating weights of the embedding matrix.

Use the function below to obtain the embedding martix for your vocabulary.

[40]:
```
def get_glove_embeddings(f_zip, f_txt, word2id, emb_size=300):

    w_emb = np.zeros((len(word2id), emb_size))

    with zipfile.ZipFile(f_zip) as z:
        with z.open(f_txt) as f:
            for line in f:
                line = line.decode('utf-8')
                word = line.split()[0]

                if word in vocab:
                    emb = np.array(line.strip('\n').split()[1:]).astype(np.
 ↪float32)
                    w_emb[word2id[word]] +=emb
    return w_emb
```

[41]:
```
w_glove = get_glove_embeddings("glove.840B.300d.zip","glove.840B.300d.
 ↪txt",vocab_word_id)
```

First, initialise the weights of your network using the `network_weights` function. Second, replace the weigths of the embedding matrix with `w_glove`. Finally, train the network by freezing the embedding weights:

[42]:
```
W = network_weights(vocab_size=len(vocab),embedding_dim=300,hidden_dim=[],␣
 ↪num_classes=3, init_val = 0.1)
#Replace the weigths of the embedding matrix with w_glove
W[0] = w_glove

for i in range(len(W)):
    print('Shape W'+str(i), W[i].shape)
```

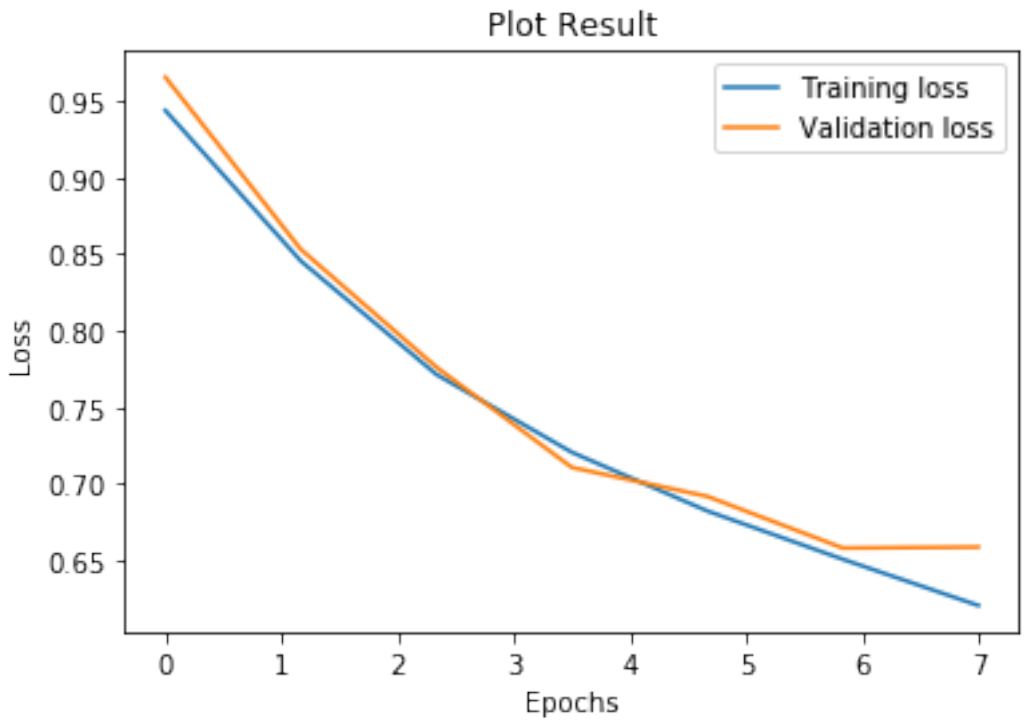
```
W, loss_tr, dev_loss = SGD(X_tr, Y_tr=x_label_tr,
                           W=W,
                           X_dev=X_dev,
                           Y_dev=x_label_dev,
                           lr=0.01,
                           dropout=0.6,
                           freeze_emb=True,
                           tolerance=0.0001,
                           epochs=30)
```

```
Shape W0 (2000, 300)
Shape W1 (300, 3)
Epoch: 0 | Training loss: 0.943680 | Validation loss: 0.965378
Epoch: 1 | Training loss: 0.845467 | Validation loss: 0.853076
Epoch: 2 | Training loss: 0.771502 | Validation loss: 0.776291
Epoch: 3 | Training loss: 0.720725 | Validation loss: 0.710855
Epoch: 4 | Training loss: 0.682446 | Validation loss: 0.691950
Epoch: 5 | Training loss: 0.650951 | Validation loss: 0.658283
Epoch: 6 | Training loss: 0.620810 | Validation loss: 0.658909
```

[43]:
```
# Now we could plot result of the training
x = np.linspace(0,len(loss_tr),len(loss_tr))
y1, y2 = loss_tr, dev_loss

plt.plot(x, y1,label='Training loss')
plt.plot(x, y2, label='Validation loss')

plt.title('Plot Result')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



[44]:
```
preds_te = [np.argmax(forward_pass(x, W, dropout_rate=0.0)['y'])+1 for x,y in␣
 ↪zip(X_test,x_label_test)]
print('Accuracy:', accuracy_score(x_label_test,preds_te))
print('Precision:', precision_score(x_label_test,preds_te,average='macro'))
print('Recall:', recall_score(x_label_test,preds_te,average='macro'))
print('F1-Score:', f1_score(x_label_test,preds_te,average='macro'))
```

```
Accuracy: 0.8720800889877642
Precision: 0.8720021421246024
Recall: 0.8720401337792643
F1-Score: 0.8719103410019038
```

5.0.1 Discuss how did you choose model hyperparameters ?

[45]:
```
# Firstly We should establish the list to select the parameters
lr_rate = [0.01,0.001]
dropout_rate = [0.4,0.5]
hyper_result = 0.0
```

```
test_result = 0.0
result = list()

for lr in range(len(lr_rate)):
    for drop in range(len(dropout_rate)):
        W =␣
↪network_weights(vocab_size=len(vocab),embedding_dim=300,hidden_dim=[],␣
↪num_classes=3, init_val = 0.1)
        W[0] = w_glove
        W, loss_tr, dev_loss = SGD(X_tr, Y_tr=x_label_tr,
                        W=W,
                        X_dev=X_dev,
                        Y_dev=x_label_dev,
                        lr=lr_rate[lr],
                        dropout=dropout_rate[drop],
                        freeze_emb=True,
                        tolerance=0.0001,
                        print_progress = False,
                        epochs=30)

        # Obtain the each result and store into different list
        # Firstly we should eveluate the performance of train
        preds_te = [np.argmax(forward_pass(x, W, dropout_rate=0.0)['y'])+1 for␣
↪x,y in zip(X_tr,x_label_tr)]
        tr_f1 = f1_score(x_label_tr,preds_te,average='macro')
        hyper_result = tr_f1

        # Obtain the test set result
        preds_te = [np.argmax(forward_pass(x, W, dropout_rate=0.0)['y'])+1 for␣
↪x,y in zip(X_test,x_label_test)]
        test_f1 = f1_score(x_label_test,preds_te,average='macro')
        test_result = test_f1

        result.append([hyper_result,test_result])
        print(" lr:"+str(lr_rate[lr])+" drop_rate:"+str(dropout_rate[drop])+"␣
↪The F1_score of Train and Test"+str(result)+"\n")
```

```
        result = list()
```

```
 lr:0.01 drop_rate:0.4 The F1_score of Train and Test[[0.8880864670764357,
0.8739796829058872]]

 lr:0.01 drop_rate:0.5 The F1_score of Train and Test[[0.8861203440327549,
0.8713124210127242]]

 lr:0.001 drop_rate:0.4 The F1_score of Train and Test[[0.863552619940834,
0.8750607645131133]]

 lr:0.001 drop_rate:0.5 The F1_score of Train and Test[[0.8626586629339302,
0.8611114954862753]]
```

6 Extend to support deeper architectures (Bonus)

Extend the network to support back-propagation for more hidden layers. You need to modify the `backward_pass` function above to compute gradients and update the weights between intermediate hidden layers. Finally, train and evaluate a network with a deeper architecture.

[46]:
```
W =␣
 ↪network_weights(vocab_size=len(vocab),embedding_dim=300,hidden_dim=[200,100],␣
 ↪num_classes=3, init_val = 0.1)
W[0] = w_glove

for i in range(len(W)):
    print('Shape W'+str(i), W[i].shape)

W, loss_tr, dev_loss = SGD(X_tr, Y_tr=x_label_tr,
                           W=W,
                           X_dev=X_dev,
                           Y_dev=x_label_dev,
                           lr=0.01,
                           dropout=0.4,
```

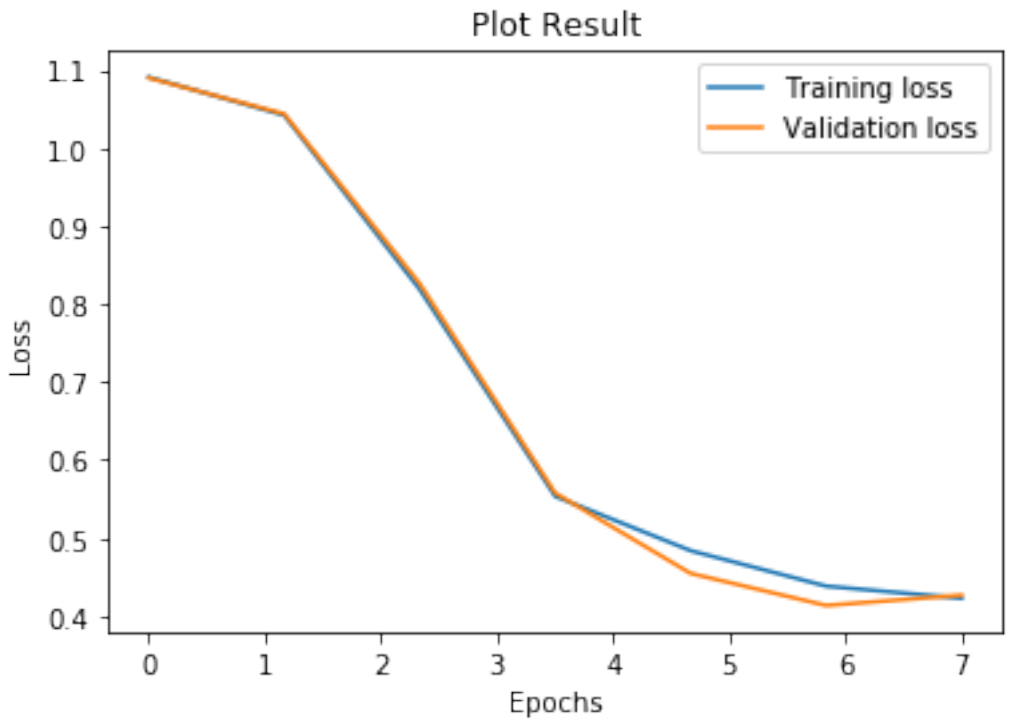
```
                                freeze_emb=True,
                                print_progress=True,
                                tolerance=0.0001,
                                epochs=30)
```

```
Shape W0 (2000, 300)
Shape W1 (300, 200)
Shape W2 (200, 100)
Shape W3 (100, 3)
Epoch: 0 | Training loss: 1.090651 | Validation loss: 1.089735
Epoch: 1 | Training loss: 1.041836 | Validation loss: 1.043621
Epoch: 2 | Training loss: 0.818768 | Validation loss: 0.826043
Epoch: 3 | Training loss: 0.553393 | Validation loss: 0.557644
Epoch: 4 | Training loss: 0.483729 | Validation loss: 0.454558
Epoch: 5 | Training loss: 0.438276 | Validation loss: 0.413482
Epoch: 6 | Training loss: 0.422741 | Validation loss: 0.426702
```

[47]:
```
# Now we could plot result of the training
x = np.linspace(0,len(loss_tr),len(loss_tr))
y1, y2 = loss_tr, dev_loss

plt.plot(x, y1,label='Training loss')
plt.plot(x, y2, label='Validation loss')

plt.title('Plot Result')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



[48]:
```
preds_te = [np.argmax(forward_pass(x, W, dropout_rate=0.0)['y'])+1 for x,y in␣
 ↪zip(X_test,x_label_test)]
print('Accuracy:', accuracy_score(x_label_test,preds_te))
print('Precision:', precision_score(x_label_test,preds_te,average='macro'))
print('Recall:', recall_score(x_label_test,preds_te,average='macro'))
print('F1-Score:', f1_score(x_label_test,preds_te,average='macro'))
```

```
Accuracy: 0.8743047830923248
Precision: 0.8757934673661216
Recall: 0.8742066146413973
F1-Score: 0.873296647575403
```

6.1 Full Results

Add your final results here:

Model	Precision	Recall	F1-Score	Accuracy
Average Embedding	0.8549264	0.8509438	0.8505701	0.8509454
Average Embedding (Pre-trained)	0.8720021	0.8720401	0.8719103	0.8720800
Average Embedding (Pre-trained) + X hidden layers (BONUS)	0.8757934	0.8742066	0.8732966	0.8743047